# Reordering Sparse Matrices for Cache-Oblivious Computations

Albert-Jan Yzelman and Rob Bisseling

2nd of June, 2010

# Outline


1. Memory and multiplication
2. Cache-friendly data structures
3. Cache-oblivious sparse matrix structure
4. Moving to two dimensions
5. Experimental results


Reference: Yzelman and Bisseling, *Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods*, SISC, 2009

# Memory and multiplication

1 Memory and multiplication

2 Cache-friendly data structures

3 Cache-oblivious sparse matrix structure

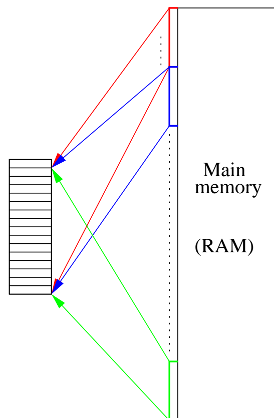
4 Moving to two dimensions

5 Experimental results

# Naive cache

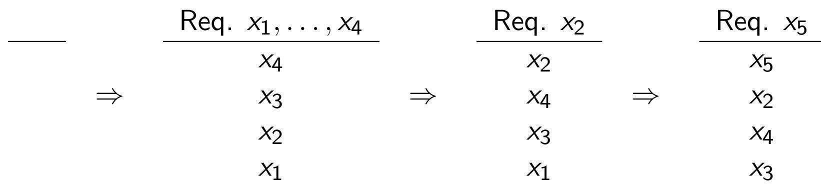
$k = 1$, modulo mapped cache

Dividing main memory (RAM) in stripes of size $L_S$, the $x$th line is mapped to the cache line number $x \bmod L$:



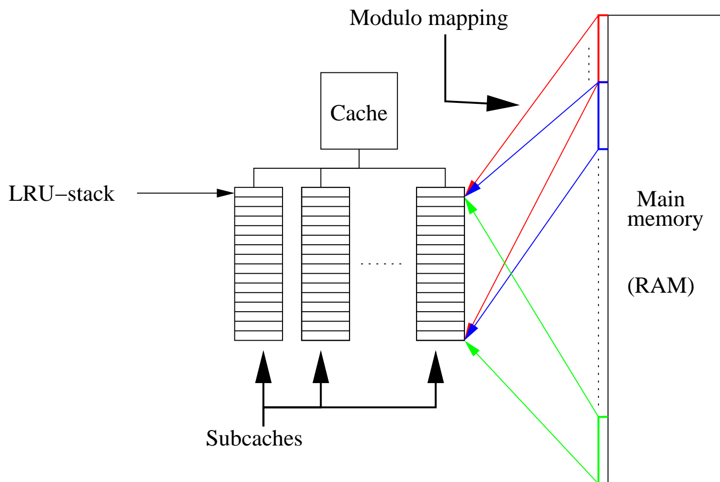
# 'Ideal' cache

Instead of a naive modulo mapping, we use a smarter policy. We take $k = L = 4$, using 'Least Recently Used (LRU)' policy:

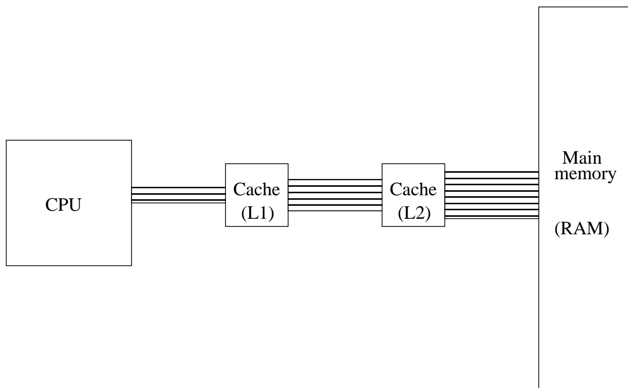


# Realistic cache

$1 < k < L$, combining modulo-mapping and the LRU policy



# Multilevel caches




| Intel Core2 (Q6600) | | | AMD Phenom II (945e) | |
|---|---|---|---|---|
| L1: | 32kB | $k = 8$ | $S = 64$kB | $k = 2$ |
| L2: | 4MB | $k = 16$ | $S = 512$kB | $k = 8$ |
| L3: | - | - | $S = 6$MB | $k = 48$ |

# The dense case

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 2$:

$x_0$

$\underline{\phantom{xxxx}} \Longrightarrow$

# The dense case

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 2$:

$$\begin{array}{c} x_0 \\ \\ \hline \end{array} \Longrightarrow \begin{array}{c} a_{00} \\ x_0 \\ \hline \end{array} \Longrightarrow$$

# The dense case

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 2$:

$$\begin{array}{c} x_0 \\ \\ \underline{\quad\quad} \end{array} \Longrightarrow \begin{array}{c} a_{00} \\ x_0 \\ \underline{\quad\quad} \end{array} \Longrightarrow \begin{array}{c} y_0 \\ a_{00} \\ \underline{x_0} \end{array} \Longrightarrow$$

# The dense case

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 2$:

$$\begin{array}{c} x_0 \\ \\ \underline{\qquad} \end{array} \Longrightarrow \begin{array}{c} a_{00} \\ x_0 \\ \underline{\qquad} \end{array} \Longrightarrow \begin{array}{c} y_0 \\ a_{00} \\ \underline{x_0} \end{array} \Longrightarrow \begin{array}{c} x_1 \\ y_0 \\ \underline{a_{00}} \\ x_0 \end{array} \Longrightarrow$$

# The dense case

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 2$:

$$\begin{array}{c} x_0 \\ \\ \hline \end{array} \Longrightarrow \begin{array}{c} a_{00} \\ x_0 \\ \hline \end{array} \Longrightarrow \begin{array}{c} y_0 \\ a_{00} \\ x_0 \\ \hline \end{array} \Longrightarrow \begin{array}{c} x_1 \\ y_0 \\ a_{00} \\ \hline x_0 \end{array} \Longrightarrow \begin{array}{c} a_{01} \\ x_1 \\ y_0 \\ \hline a_{00} \\ x_0 \end{array} \Longrightarrow$$

# The dense case

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 2$:

$$\begin{array}{c} x_0 \\ \\ \hline \end{array} \Longrightarrow \begin{array}{c} a_{00} \\ x_0 \\ \hline \end{array} \Longrightarrow \begin{array}{c} y_0 \\ a_{00} \\ x_0 \\ \hline \end{array} \Longrightarrow \begin{array}{c} x_1 \\ y_0 \\ a_{00} \\ \hline x_0 \end{array} \Longrightarrow \begin{array}{c} a_{01} \\ x_1 \\ y_0 \\ \hline a_{00} \\ x_0 \end{array} \Longrightarrow \begin{array}{c} y_0 \\ a_{01} \\ x_1 \\ \hline a_{00} \\ x_0 \end{array}$$

When $k, L$ are a bit larger, we can predict the following:

- the lower elements from the vector $x$ (that is, $x_0, x_1, \ldots, x_i$ for some $i < n$) are evicted while processing the entire first row. This causes $\mathcal{O}(n)$ cache misses on the remaining $m - 1$ rows.

Fix:

- stop processing a row before an element from $x$ would be evicted and first continue row-wise: i.e., process $Ax$ by doing MVs on $m \times q$ submatrices: $y = a_0 x + a_1 x + \ldots$

When $k, L$ are a bit larger, we can predict the following:

- the lower elements from the vector $x$ (that is, $x_0, x_1, \ldots, x_i$ for some $i < n$) are evicted while processing the entire first row. This causes $\mathcal{O}(n)$ cache misses on the remaining $m - 1$ rows.

Fix:

- stop processing a row before an element from $x$ would be evicted and first continue row-wise: i.e., process $Ax$ by doing MVs on $m \times q$ submatrices: $y = a_0x + a_1x + \ldots$

Unwanted side effect:

- now lower elements from the vector $y$ can be prematurely evicted...

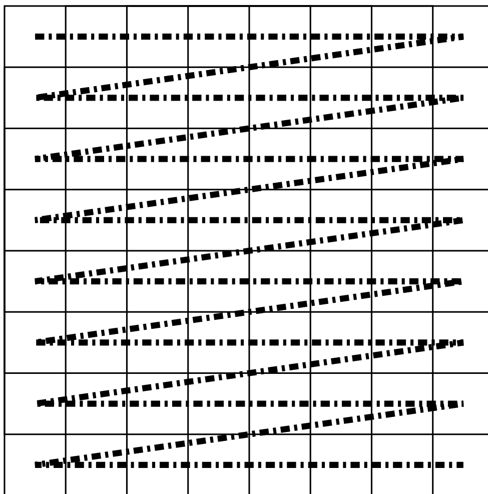
Fix:

- stop processing a submatrix before an element from $y$ would be evicted; the MV routine now is applied on $p \times q$ submatrices.

This approach is *cache-aware*; implemented in, e.g., GotoBLAS.

# The sparse case

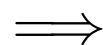
Standard datastructure: Compressed Row Storage (CRS)



# The sparse case

Sparse matrix–vector multiplication (SpMV)

$x_?$



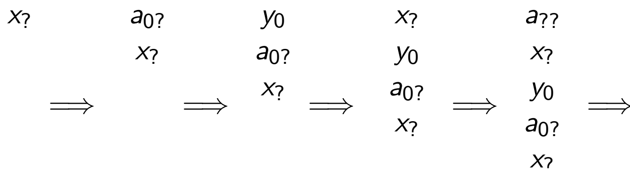
# The sparse case

Sparse matrix–vector multiplication (SpMV)

$x_?$ $\Longrightarrow$ $a_{0?}$ $x_?$ $\Longrightarrow$ $y_0$ $a_{0?}$ $x_?$ $\Longrightarrow$

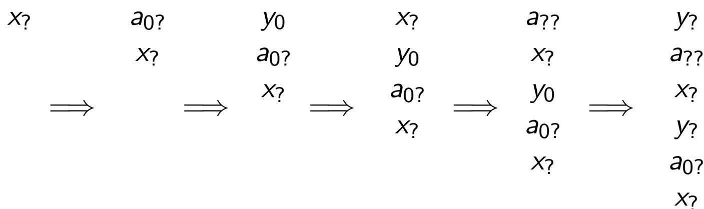
# The sparse case

Sparse matrix–vector multiplication (SpMV)



# The sparse case

Sparse matrix–vector multiplication (SpMV)




We cannot predict memory accesses in the sparse case!

# Cache-friendly data structures

1 Memory and multiplication

2 Cache-friendly data structures

3 Cache-oblivious sparse matrix structure

4 Moving to two dimensions

5 Experimental results

# CRS

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$

Stored as:

nzs: [4 1 3 2 3 1 2 7 1 1]
col: [0 1 2 2 3 0 3 0 2 3] , $2nnz + (m + 1)$ accesses
row: [0 3 5 7 10]

# Incremental CRS

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$

Stored as:

nzs: [4 1 3 2 3 1 2 7 1 1]
col_increment: [0 1 1 4 1 1 3 1 2 1] , $2nnz + m$ accesses
row_increment: [0 1 1 1]

Note: accesses like plain CRS, but requires less instructions for SpMV

Reference: Joris Koster, *Parallel templates for numerical linear algebra, a high-performance computation library* (Masters Thesis), 2002.

# Blocked CRS

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}, \text{ dense blocks: } 4,1,3 \;/\; 2,3 \;/\; 1 \;/\; 2 \;/\; 7,0,1,1$$

Stored as:

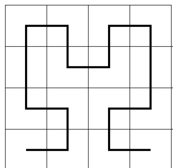
nzs: [4 1 3 2 3 1 2 7 0 1 1]
blk: [0 3 5 6 7 11]
col: [0 2 0 3 0]
row: [0 1 2 4 5]

, $nnz + (2nblk + 1) + (m + 1)$ accesses

Reference: Pinar and Heath, *Improving Performance of Sparse Matrix-Vector Multiplication*, 1999

# Fractal datastructures (triplets)

$$A = \begin{pmatrix} 4 & 1 & 0 & 2 \\ 0 & 2 & 0 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 0 \end{pmatrix}$$



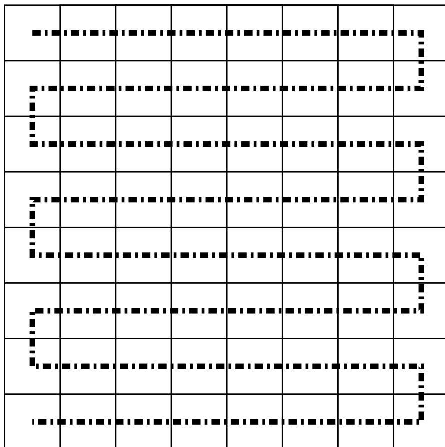
Stored as:

nzs: [7 1 4 1 2 2 3 2 1]
i : [3 2 0 0 1 0 1 2 3] , **3nnz** accesses per nonzero
j : [0 0 0 1 1 3 3 3 2]

Reference: Haase, Liebmann and Plank, *A Hilbert-Order Multiplication Scheme for Unstructured Sparse Matrices*, 2005

# Zig-zag CRS

Change the order of CRS:



# Zig-zag CRS

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$

Stored as:

nzs: [4 1 3 3 2 1 2 1 1 7]
col: [0 1 2 3 2 0 3 3 2 0] , $2nnz + (m + 1)$ accesses
row: [0 3 5 7 10]

Reference: Yzelman and Bisseling, *Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods*, SISC, 2009

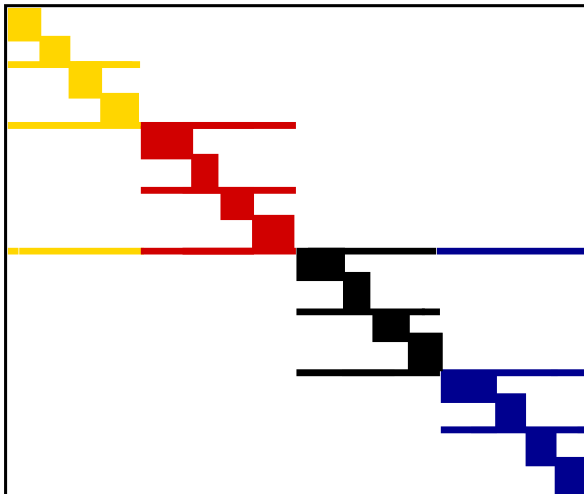
# Cache-oblivious sparse matrix structure


1. Memory and multiplication
2. Cache-friendly data structures
3. Cache-oblivious sparse matrix structure
4. Moving to two dimensions
5. Experimental results

## Why not also change the input matrix structure?

- Assuming zig-zag CRS ordering (theoretically)
- Allowing only row and column permutations

# Separated Block Diagonal form



# Separated Block Diagonal form

No cache misses

1 cache miss per row

3 cache misses per row

1 cache miss per row

# Separated Block Diagonal form

No cache misses

1 cache miss per row

3 cache misses

1 cache miss per row

7 cache misses per row

1 cache miss per row

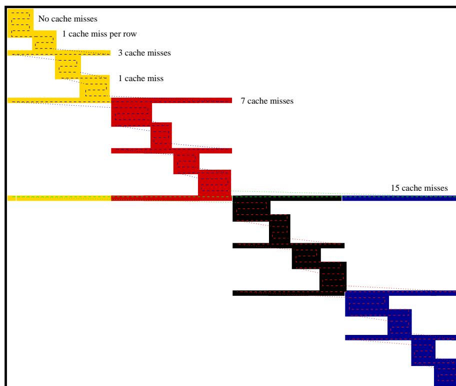
3 cache misses per row

1 cache miss per row

Assuming a row-net hypergraph representation $(\mathcal{V}, \mathcal{N})$, we will have partitioned $\mathcal{V}$ into subsets $\mathcal{V}_0, \ldots, \mathcal{V}_{p-1}$. Nets spread over multiple parts correspond to mixed-row blocks incurring cache misses.

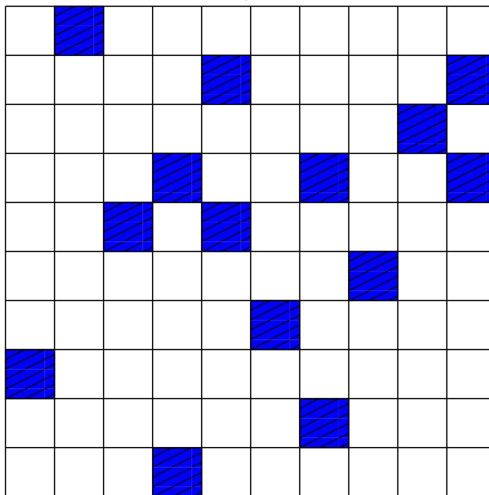
No cache misses
1 cache miss per row
3 cache misses
1 cache miss
7 cache misses
15 cache misses

# Separated Block Diagonal form



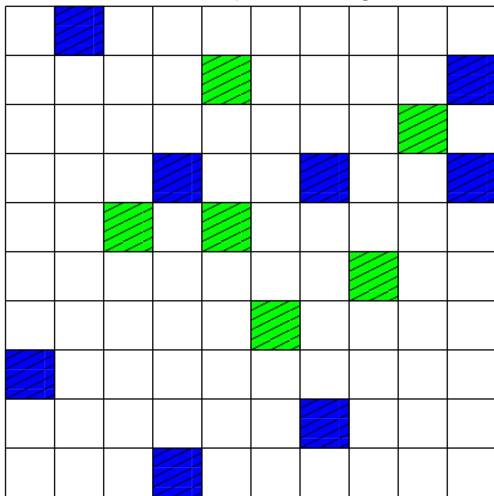

(Upper bound on) the number of cache misses: $\sum_i (\lambda_i - 1)$

# Permuting to SBD form



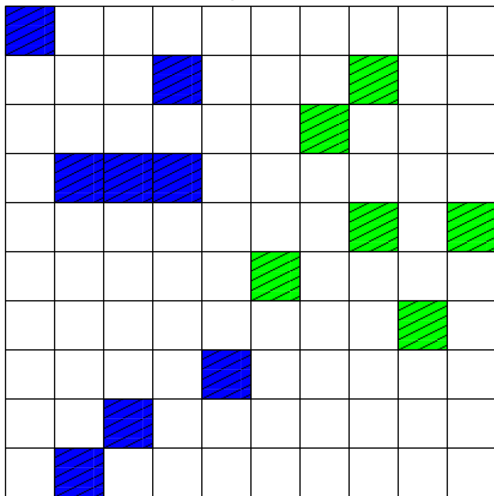
# Permuting to SBD form

Column partitioning



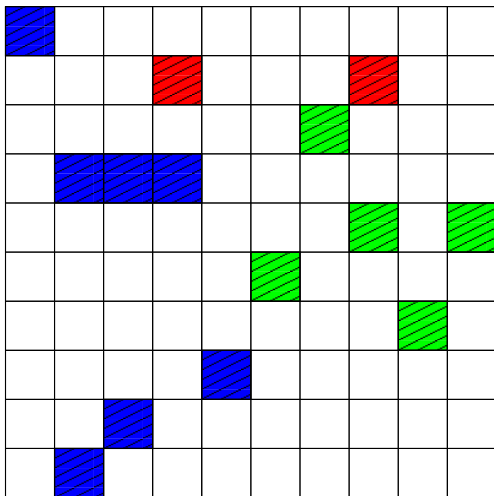
# Permuting to SBD form

Column permutation



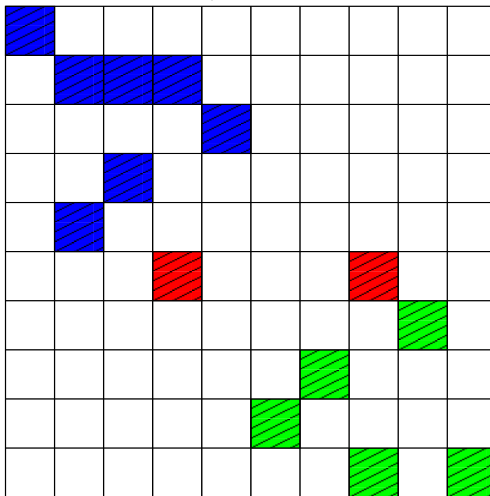
# Permuting to SBD form

Mixed row detection

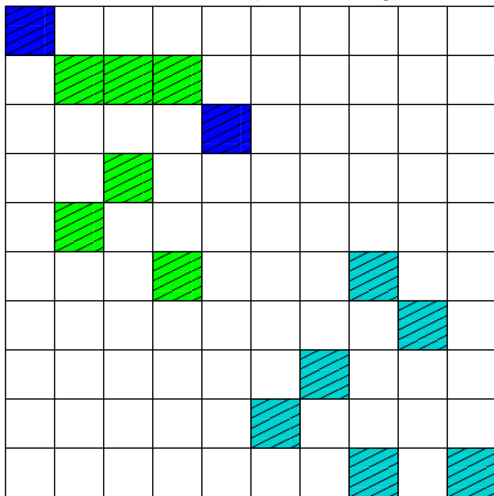


# Permuting to SBD form

Row permutation

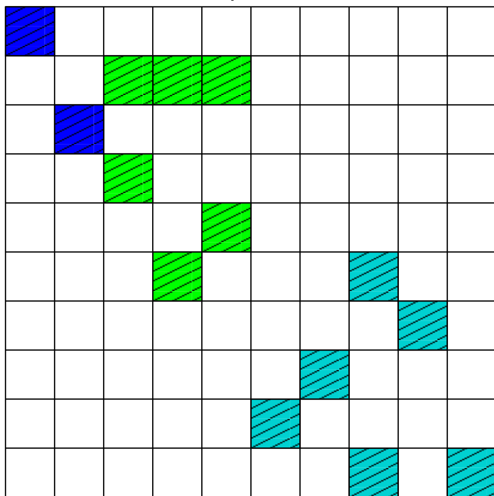


# Permuting to SBD form



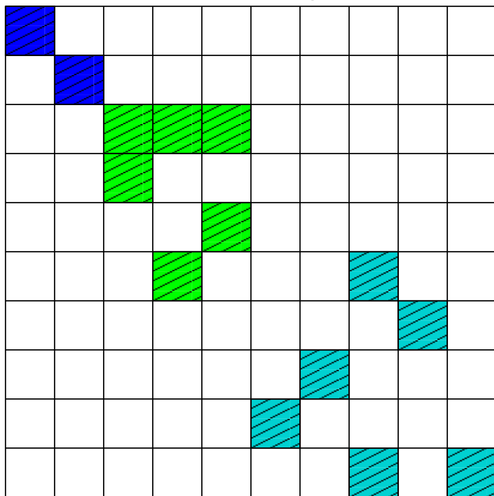
# Permuting to SBD form

Column permutation



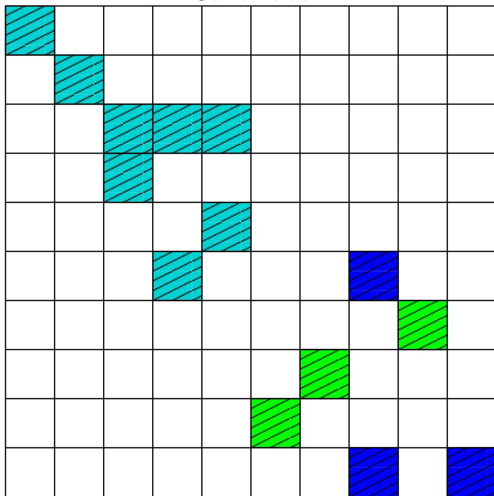
# Permuting to SBD form

No mixed rows - row permutation

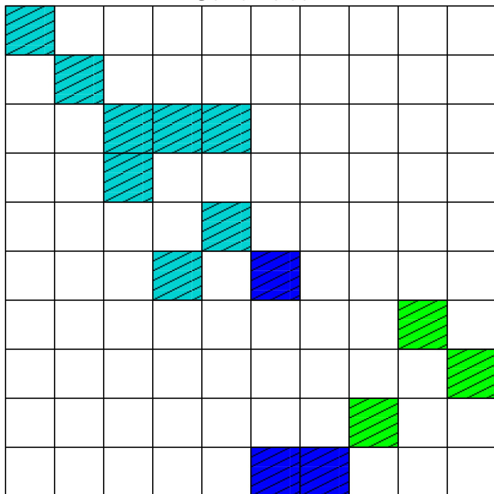


# Permuting to SBD form

Continued

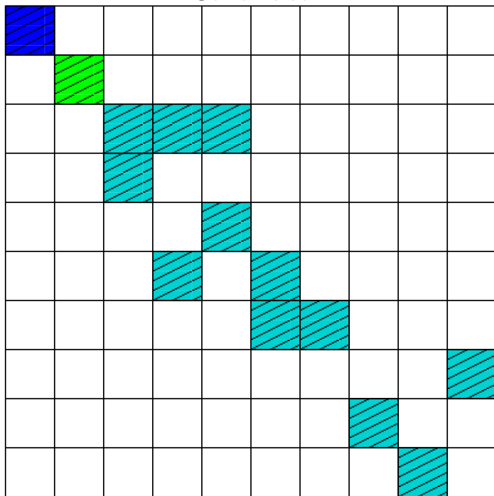


# Permuting to SBD form

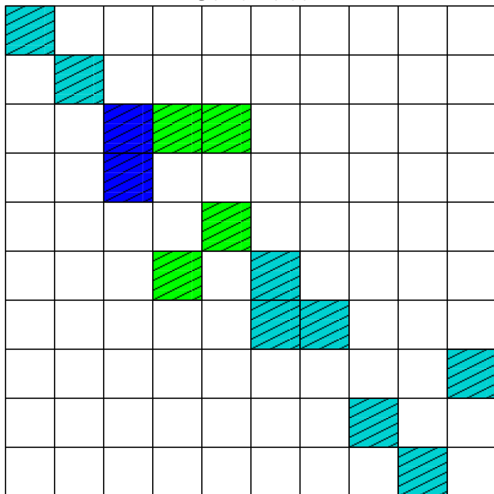


# Permuting to SBD form

Continued

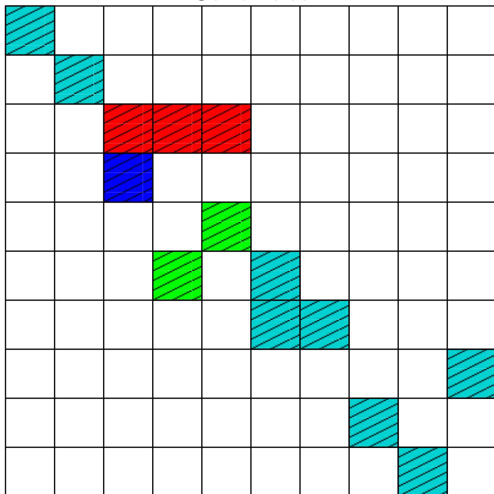


# Permuting to SBD form



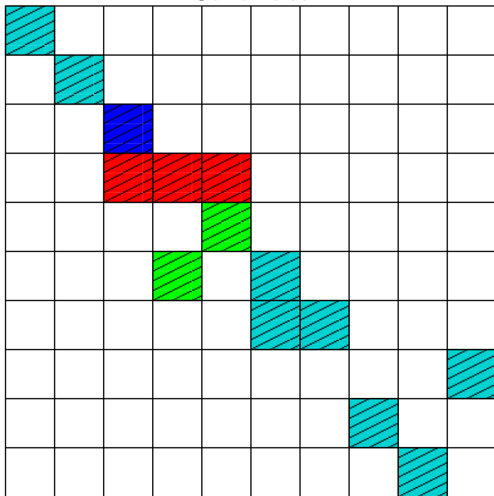
# Permuting to SBD form

Continued

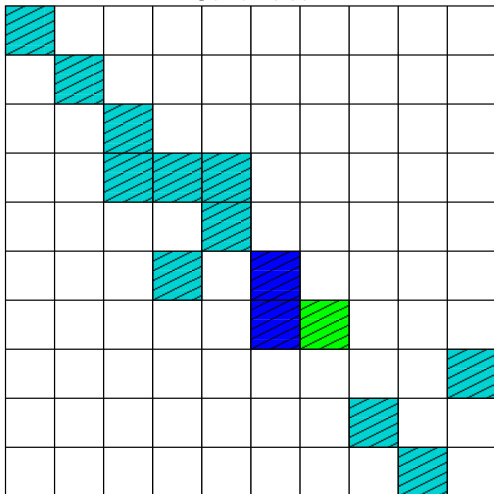


# Permuting to SBD form

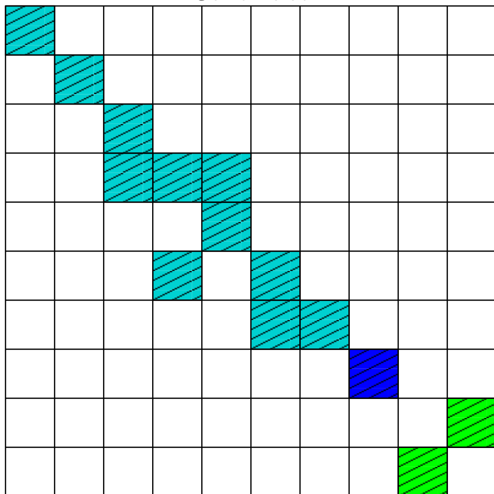
Continued



# Permuting to SBD form

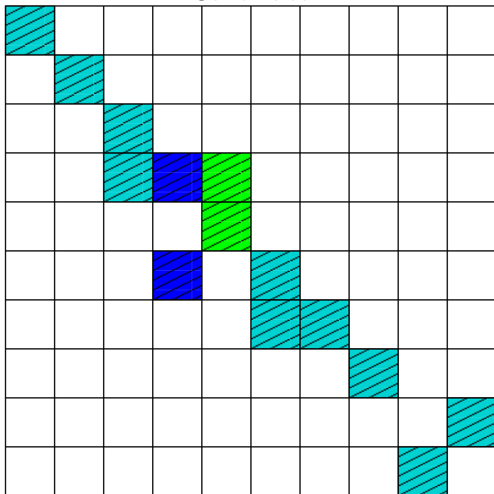


# Permuting to SBD form



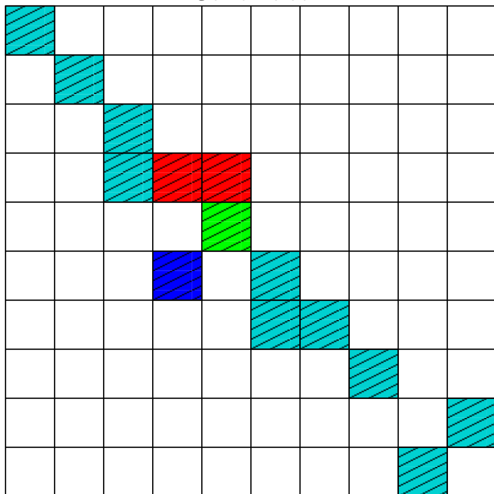
# Permuting to SBD form

Continued



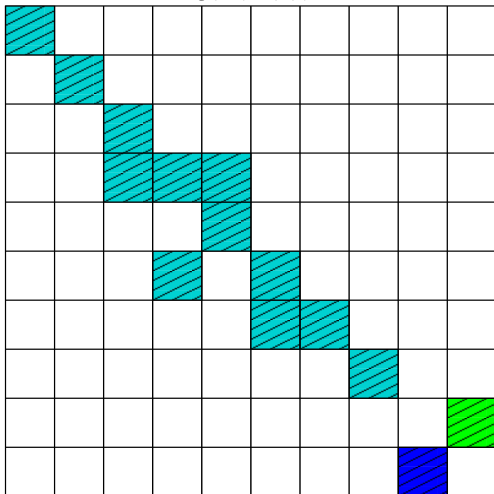
# Permuting to SBD form

Continued

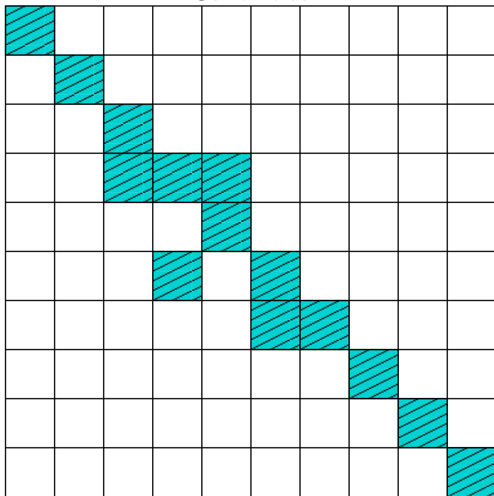


# Permuting to SBD form

Continued



# Permuting to SBD form



For $p = \frac{n}{wL}$, the number of cache misses is given by

$$\sum_{i:\ n_i \in \mathcal{N}} (\lambda_i - 1), \quad \text{assuming Zig-zag CRS.}$$

This is also called the $\lambda - 1$ *metric*, and is already used extensively in parallel computing. Partitioners should also take into account the *load-imbalance*, typically denoted by $\epsilon$.

References:

- Çatalyürek and Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, 1999
- Vastenhouw and Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, 2005

For a truly cache-oblivious approach, the number of subsets $p$ we partition $\mathcal{V}$ into, must be as large as possible:

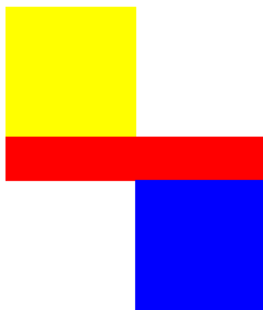
$$p \to \infty \quad (p = n).$$

Taking $p$ higher than needed for the actual cache size does not harm efficiency, and will even optimise for the smaller caches closer to the CPU; *we optimise access for all cache levels irrespective of the cache parameters*.

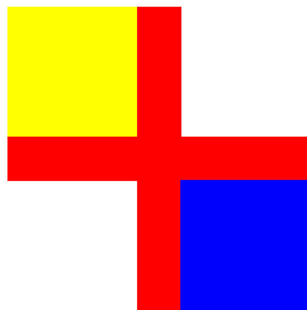
# Moving to two dimensions

1. Memory and multiplication
2. Cache-friendly data structures
3. Cache-oblivious sparse matrix structure
4. Moving to two dimensions
5. Experimental results

# Two-dimensional SBD



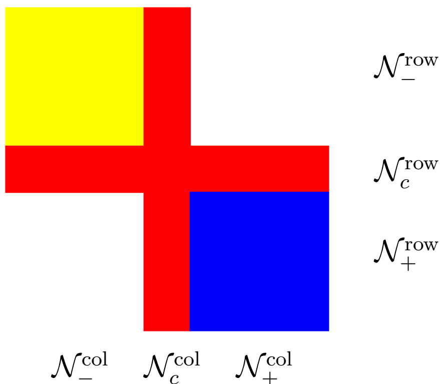
**1D**



**2D**

# Two-dimensional SBD

Using a fine-grain model of the input sparse matrix, individual nonzeros each correspond to a vertex; each row and column have a corresponding net.
The quantity to minimise remains $\sum_i(\lambda_i - 1)$.



# Two-dimensional SBD

Zig-zag CRS is not suitable for handling 2D SBD

# Two-dimensional SBD

# Two-dimensional SBD

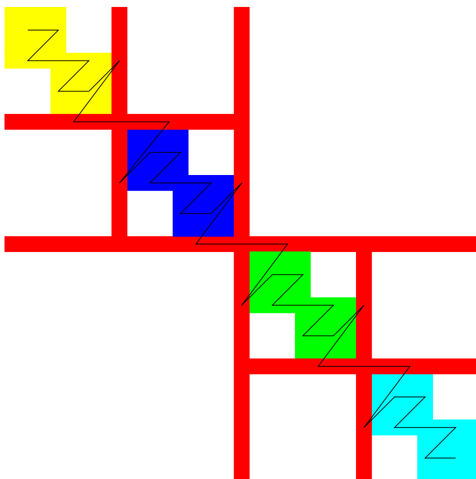
# Bi-directional Incremental CRS (BICRS)

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$

Stored as:

nzs: [3 2 3 1 1 2 1 7 4 1]
col_increment: [2 4 1 4 -1 5 -3 4 4 1] ,
row_increment: [0 1 2 -1 1 -3]

$2nnz + (row_jumps + 1)$ accesses.

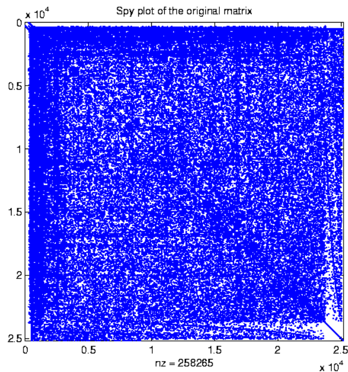


This means storing each sparse block should be done in one of the CRS orderings (CRS/ICRS/ZZ-CRS/ZZ-ICRS).

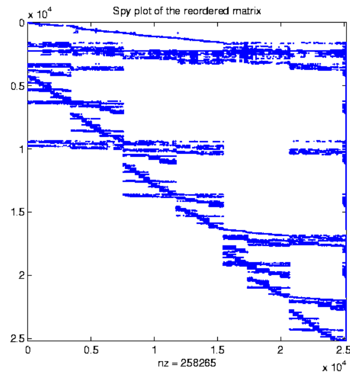
# Experimental results


1. Memory and multiplication
2. Cache-friendly data structures
3. Cache-oblivious sparse matrix structure
4. Moving to two dimensions
5. Experimental results

# The rhpentium matrix – 1D SBD




$p = 1$ (original)

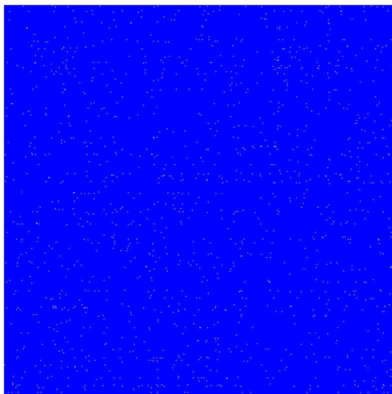



$p = 100, \quad \epsilon = 0.1$

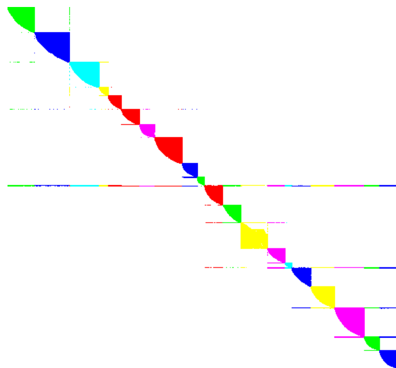
# The rhpentium matrix – 2D SBD

$p = 100, \quad \epsilon = 0.1$

# The Stanford link matrix

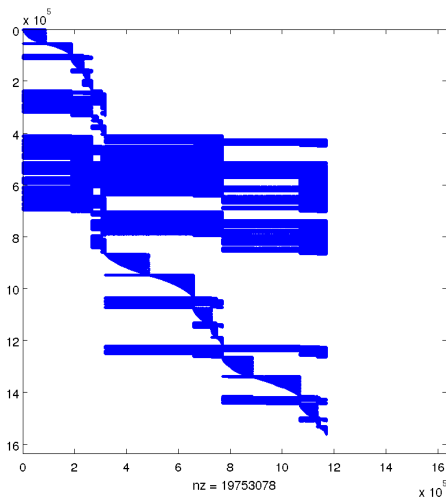


Original



Finegrain (hybrid, $p = 20$, $\epsilon = 0.1$)

# The wikipedia matrix (May 11, 2005)




With $p = 20$, $\epsilon = 0.1$.

Real–time speedup after cache–oblivious reordering

Run–time ratio

1.1
1.05
1
0.95
0.9
0.85
0.8
0.75

1 2 3 4 5

Matrix number

Original
2/0.1
3
4
100
400
2/0.3
3
4
100
400

Matrices used are: 1. memplus, 2. rhpentium, 3. s3dkt3m2, 4. rand10000, and 5. fidap037.

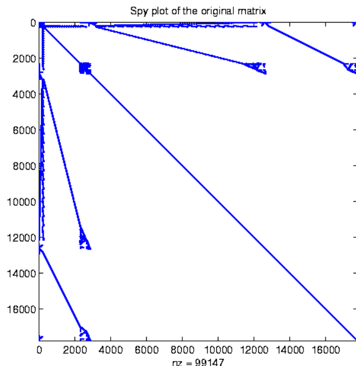
Real–time speedup after cache–oblivious reordering

| Matrix number | Original | 2/0.1 | 3 | 4 | 100 | 400 | 2/0.3 | 3 | 4 | 100 | 400 |
|---|---|---|---|---|---|---|---|---|---|---|---|

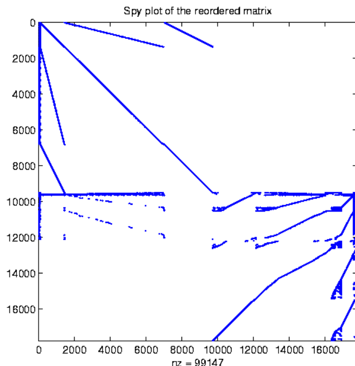


Matrices used are: 6. lhr34; 7. rand50000; 8. nug30; 9. tbdlinux; 10. bmw7st1.

# The memplus matrix – 1D SBD

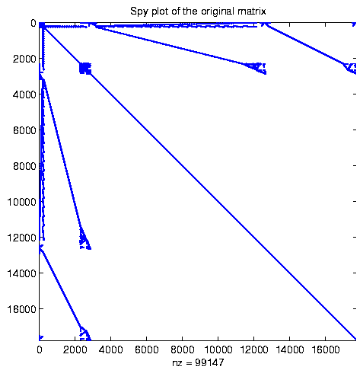



$p = 1$ (original)

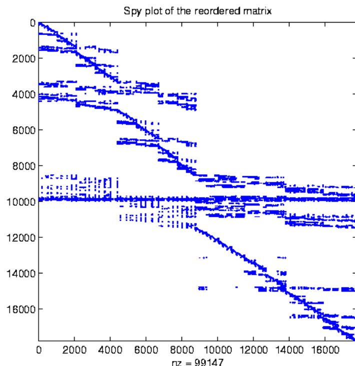



$p = 2, \quad \epsilon = 0.1$

# The memplus matrix – 1D SBD

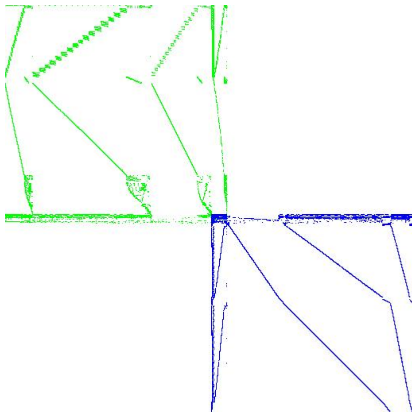



$p = 1$ (original)

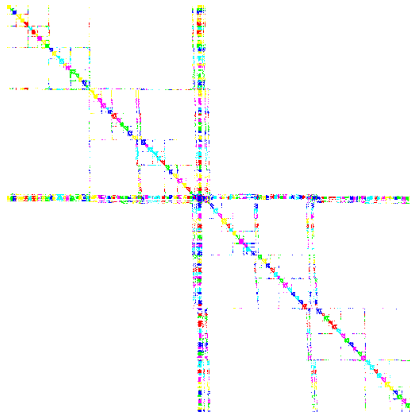



$p = 100, \quad \epsilon = 0.1$

# The memplus matrix – 2D SBD



$p = 2$



$p = 100$

Real–time speedup after cache–oblivious reordering

Run–time ratio: 1.3, 1.2, 1.1, 1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3

Matrix number: 11, 12, 13, 14

Legend: CRS, 1, 2, 3, 4, 5, 10, 15, 20

Matrices used are: 11. Stanford, 12. Stanford_Berkeley, 13. wikipedia20051105, and 14. cage14.

# Pre-processing times

## 1D

| Matrix | Reordering time | SpMV time (old/new) |
|---:|---|---:|
| rhpentium, $p = 400$: | 1 minute | (0.9 / 0.7 ms.) |
| memplus, $p = 100$: | 4 minutes | (0.4 / 0.4 ms.) |
| stanford, $p = 20$: | 4 minutes | (27 / 14 ms.) |
| cage14, $p = 20$: | 12 minutes | (109 / 123 ms.) |
| stanford_berkeley, $p = 20$: | 13 minutes | (31 / 30 ms.) |
| wikipedia, $p = 10$: | 16 minutes | (353 / 236 ms.) |
| wikipedia, $p = 20$: | 45 minutes | (353 / 237 ms.) |

Results from 2009, run on an Intel Q6600.

| Name | $p=2$ | $p=3$ | $p=4$ | $p=100$ | $p=400$ |
|---|---|---|---|---|---|
| memplus | 1531 | 1914 | 1818 | 12793 | 101744 |
| rhpentium | 5090 | 6752 | 7303 | 17064 | 60251 |
| s3dkt3m2 | 603 | 673 | 740 | 1934 | 5181 |
| rand10000 | 1560 | 1411 | 1820 | 23179 | 103565 |
| fidap037 | 1005 | 1068 | 1131 | 5657 | 12761 |

| Name | $p=2$ | $p=3$ | $p=4$ | $p=10$ | $p=20$ |
|---|---|---|---|---|---|
| stanford | 5139 | 7603 | 6828 | 7922 | 8332 |
| stanford_berkeley | 11305 | 13208 | 15093 | 19138 | 21260 |
| wikipedia20051105 | 2152 | 1992 | 2168 | 2570 | 7418 |
| cage14 | 4238 | 3987 | 3635 | 4583 | 5611 |

Table: Cost of reordering in terms of the number matrix multiplications on the original matrix. Here, $\epsilon = 0.1$. Construction times were measured on an Intel Core 2 (Q6600) machine. These figures are from 2009.

# Pre-processing times

2D, *(preliminary)*

| Matrix | Reordering time | SpMV ratio (new/old) |
|---|---|---|
| rhpentium, $p = 2$: | $<$ 1 minute | 1.01 (ICRS/ICRS) |
| rhpentium, $p = 2$: | $<$ 1 minute | 0.88 (BICRS/BICRS) |
| stanford, $p = 20$: | 8 minutes | 0.44 (BICRS/ICRS) |
| wikipedia, $p = 20$: | 17 minutes | 0.83 (BICRS/BICRS) |

*Old*: SpMV on the original matrix $A$
*New*: SpMV on the reordered matrix $PAQ$

Reordering done on Huygens (IBM Power6), SpMVs on an Intel Atom. Data from February 2010.

*BICRS nonzero ordering is only partially implemented*

# Conclusions:

we have introduced a scheme capable of increasing SpMV performance up to a factor two, while:

- remaining cache-oblivious, and
- keeping open the option of using specialised sparse BLAS libraries (1D).

For already well-structured matrices our approach does not obtain significant speedups, but does not decrease performance much either.

The two-dimensional method promises even greater speedups, more than doubling SpMV speed on the Stanford link matrix in its current preliminary version.

# Near-future advances

- a new article on the full 2D reordering method is underway;
- Mondriaan 3.0, which will include these reordering methods, is close to release;
- new research, in collaboration with Madan Sathe (Univ. of Basel), on a different applications of partion-based reordering is forthcoming.